Nearest-Neighbor Caching for Content-Match Applications

Sandeep Pandey[1] Andrei Broder[1] Flavio Chierichetti[2*]
Vanja Josifovski[1] Ravi Kumar[1] Sergei Vassilvitskii[1]

[1]Yahoo! Research
701 First Ave.
Sunnyvale, CA 94089.
{spandey,broder,vanjaj,ravikumar,sergei}@yahoo-inc.com

[2]Dipartimento di Informatica
Sapienza University of Rome
Roma, 00198, Italy
chierichetti@di.uniroma1.it

ABSTRACT

Motivated by contextual advertising systems and other web applications involving efficiency–accuracy tradeoffs, we study similarity caching. Here, a cache hit is said to occur if the requested item is similar but not necessarily equal to some cached item. We study two objectives that dictate the efficiency–accuracy tradeoff and provide our caching policies for these objectives. By conducting extensive experiments on real data we show similarity caching can significantly improve the efficiency of contextual advertising systems, with minimal impact on accuracy. Inspired by the above, we propose a simple generative model that embodies two fundamental characteristics of page requests arriving to advertising systems, namely, long-range dependences and similarities. We provide theoretical bounds on the gains of similarity caching in this model and demonstrate these gains empirically by fitting the actual data to the model.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Performance, Experimentation

Keywords

Caching, nearest-neighbor search, content-match

1. INTRODUCTION

Consider an internet advertising system tasked with showing contextual advertisements (ads) on a publisher's page when a user visits the page. This so-called *content-match* system strives to choose and serve the best ads using a range of cues — the profile of the user and the content on the publisher's page — to search the pool of available ads and retrieve one or more ads to display on the page. The challenge is to accomplish this task as efficiently as possible, given hundreds of millions of available ads. Since the click-through rate on contextual ads is lower than sponsored-search ads, the revenue per serving is lower, and this applies even more pressure on lowering the cost of serving contextual ads.

A natural way would be to cast the ad serving problem as nearest-neighbor search: treat the user profile and the page content as vectors in a high-dimensional space, preprocess each ad to map it to the same space, and develop a notion of similarity between a user-page vector and an ad vector in this space. At this juncture, it becomes important to draw a comparison between this scenario and the traditional keyword-based web search. While both deal with a large amount of data, they differ at a fundamental level. First, content-match has a higher volume of traffic for the simple reason that people browse more than they search. Second, the latency requirements for content-match are stricter than for web search — the additional time to load the contextual ads should be barely noticeable when a user loads a publisher's page in her browser. Third, web search engines manage latency by heavy use of caching, from caching the posting lists all the way to result set caching; this is crucial since the query distribution is heavy-tailed. It is not clear if caching will particularly benefit a content-match system since the each user-page vector can be almost unique. Fourth, inverted index, which forms the core of keyword-based web, is far more well-understood and scalable data structure than those that exist for the more challenging nearest-neighbor search. And fifth, the half-life of ads is much smaller than that of web pages and this adds to the complexity of building a scalable ad serving system.

On the other hand, the problem of selecting and serving contextual ads offers the following flexibility: similar ads can be shown to similar users visiting similar pages. In other words, the nearest-neighbor search need not be exact and it is acceptable to provide an approximate answer to the user-page vector. There are two ways of exploiting this flexibility. The first is to develop fast approximate nearest-neighbor search algorithms; however, it is not immediate if these algorithms will still be scalable in the vast world of ads. The second way, inspired by the web search setting, is to develop caching policies that can judiciously take advantage of the freedom to provide an approximate answer; this is the focus of our work. We study policies for similarity caching [6], where a cache hit is said to occur if the requested item is similar but not necessarily equal to some cached item. Conventional caching (i.e., exact caching) is not very effective in such approximate nearest-neighbor search scenarios, as shown by our experiments on real data gathered from a large content match application, because: (a) user-page

*This work was done while the author was visiting Yahoo! Research.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

request vectors tend to be long and unique making exact caching ineffective, and (b) each cache item is quite large (as explained in Section 4) limiting the size and coverage of the cache.

A moment of reflection distills the basic characteristic of the above scenario: achieving an efficiency–accuracy tradeoff. Such tradeoffs are quite common in many web applications. We mention two such applications. The first is personalized search. Here, the goal is tailor the web search results to a user depending on her profile. While it is prohibitive to compute a per-user ranking for each query, it is yet reasonable to assume similar users can be shown similar search results. The second application is in content-based image search, where it may suffice to show a cached image that is similar to a query image; independent of our work, Falchi et al. [9] recently studied similarity caching in this context.

Our contributions. In this paper we study similarity caching in content-match systems. We study two objectives that dictate the efficiency–accuracy tradeoff. In the first objective, a cache hit is said to occur if the similarity between two items is more than a pre-specified threshold. Here, we propose generalizations of two well-known algorithms in the classical caching setting: least recently used (LRU) and least frequently used (LFU). Our generalizations go beyond just redefining what a cache hit means. They make full use of the wiggle room available by ensuring that the cached items cover the similarity space without much overlap. In the second objective, there is a smooth tradeoff between the IO cost and utility of offering an item similar (but not equal) to the requested item. Here, we propose a simple randomized policy that generalizes both LRU and LFU. We conduct extensive experiments on data from a real content-match system. Our experiments show that similarity caching can significantly improve the performance of content-match systems, without compromising much in terms of accuracy.

Inspired by the patterns observed in the data and the performance of the similarity caching algorithms, we propose a simple generative model for content-match requests. This model captures two fundamental characteristics of content-match requests: long-range dependences (user-page visits have a heavy tail) and similarities (presence of many similar users and similar pages). In this model, we give a theoretical upper bound on the performance of classical caching and a lower bound on the performance of similarity caching. We compute the parameters of this model from the content-match data using the maximum-likelihood principle and show that the actual performance numbers agree with our theoretical estimates.

Organization. The paper proceeds as follows. Section 2 overviews the related work. Sections 3 describes various similarity caching objectives and policies. Section 4 contains the experimental evaluation. Section 5 presents a simple request arrival model, whose parameters can be obtained from real data by maximum likelihood, and shows the quantitative gain of similarity caching over exact caching. Section 6 contains concluding remarks.

2. RELATED WORK

Caching has long been recognized as a critical component in high performance applications. Indeed, most applications utilize caching on many levels, and multi-level schemes have recently been studied [16]. The closest form of caching to this work is *result caching.* In this framework the set of results for popular queries is fetched from the cache, rather than being recomputed every time the query is issued. In web search applications result caching has been extensively studied, see, for example, [8, 14, 15, 17, 19]; and it is safe to say that all modern search engines use some form of result caching. In the theoretical community, this kind of a caching problem serves as a cornerstone of competitive analysis theory. The well known caching schemes, e.g. LRU, LFU, have been analyzed and the limits on their performance are well understood [18].

At first glance, similarity caching may look like a variation on a clustering problem – the goal is to find a set of points that best represent the incoming queries. What makes this problem different is the online component – while the clusters must remain relatively stable in order to save on the computation costs, the clusters must also evolve as the query distribution changes. Capturing this trade-off is at the heart of this work. Independent of our work, Falchi et al [9] recently introduced a notion of a *metric cache*, and showed an increase in the hit rate when using a simple variant of LRU. In this work, we derive adaptive caching strategies, demonstrate their efficacy on real datasets and prove a bound on the expected increase in performance due to similarity caching.

On the theoretical side, the problem can be formulated as a special case of the so-called Metrical Task Systems (MTS) problem, see [2] for a full description. However, MTS is a much more general problem, and, as such, the known algorithms achieve very weak performance guarantees. Recently Chierichetti et al. [6] provided the first theoretical analysis of similarity caching. They show that the competitive ratio of *any* caching scheme depends crucially on the dimensionality of the space and complement their analysis with several algorithms for similarity caching when the points lie in Euclidean space. While providing an interesting analysis, their results do not extend to the Jaccard similarity model that we consider in this work, nor do they consider the nearest-neighbor search problem that comes up in finding the closest point in the cache.

Solving the nearest neighbor problem efficiently is a keystone of an effective similarity caching strategy. The problem has a rich history, see for example the survey by Indyk [12], and the references therein. In this work we use Locality Sensitive Hashing [7, 13] to solve the nearest neighbor problem, and demonstrate its effectiveness against exhaustive search.

3. SIMILARITY CACHING

We first briefly describe the basic principles behind caching. To avoid ambiguity, we use *exact caching* to refer to the classical notion. We then describe the similarity caching setup, including the objectives, policies, and practical considerations. In our discussions, a request item p is always associated with a key-value pair $\langle \mathrm{key}(p), \mathrm{val}(p)\rangle$; the requested item is specified by $\mathrm{key}(p)$, and $\mathrm{val}(p)$ is returned as the result. Typically, the size of the key is insignificant compared to that of the value, i.e., $|\mathrm{key}(p)| \ll |\mathrm{val}(p)|$. In our content-match application for instance, the size of key is roughly 1KB while the size of the value can be 10s of MBs.

3.1 Caching basics

An exact caching scheme works in the following way. On

receiving a request for an item p specified by key(p), the cache is probed to check if it has p. If so, then this is called a *hit* and the cached item is used to serve the request. If p is not found in the cache, then this is called a *miss* and p is brought into the cache from the disk. If the cache is full, then an existing cached item is *evicted* to make space for p. The main objective in exact caching is to serve as many requests from the (fixed-size) cache as possible, i.e., maximize the number of cache hits or minimize the number of cache misses (disk accesses). Thus, exact caching consists of two steps, namely, hit-or-miss determination and an eviction policy.

While the hit-or-miss determination is trivial in exact caching, the eviction policy can be realized in one of several ways. We briefly recount two popular ways: least recently used (LRU) and least frequently used (LFU). The LRU policy exploits the temporal locality in a request stream, i.e., recent requests are likely to be re-requested in the near future. To implement this, LRU associates with each item in the cache a reference time that denotes the most recent moment when this item was used to serve a request. When needed, LRU evicts the cache item with the least reference time.

The LFU policy keeps counts of how many times each cache item was hit in the past and evicts the least frequently used item when needed. Several variants of LFU have been proposed to deal with its different shortcomings [1, 21]. For example, it may happen in LFU that certain items occur in a burst to accumulate such high frequency counts that they never get evicted from the cache. Window-LFU deals with this by counting the frequency of each item within a recent finite length time-window.

3.2 Caching with similarity

As discussed earlier, there are several web applications where the concept of exact caching can be relaxed, leading to similarity caching. The goal now is to serve items with keys that are "similar" enough to the keys of the requested items with as few cache misses as possible; we assume a notion of similarity that can be defined between the keys of the items. There is a clear tradeoff between the similarity of offered items to the requested items and the incurred (disk access) cost.

We revisit the caching basics in this context. To begin with, the semantics of a hit-or-miss is not clear in similarity caching — how similar should two item keys be that we can declare a cache hit? And operationally, how to check if a similar item indeed exists in the cache? It is clearly expensive to perform an exhaustive search of the cache to look for the item most similar to the requested item.

Likewise, the rest of the caching steps can potentially exploit the flexibilities offered by similarity caching. For example, it might be beneficial to bring a requested item into the cache even for a cache hit (under similarity). And, it might make sense to incrementally "re-organize" the cache to ensure that cached items are "well-separated" in the similarity space. This takes full advantage of similarity and enables us to work with a smaller cache.

Objective. More formally, we can define our similarity caching problem as follows: let b denote the average IO cost budget, i.e., b is the fraction of cache misses. Let $c(p)$ denote the (cached) item offered by the caching policy for request p. Then for a given cache size, an IO budget B, and a list of requests (specified by keys), say $\mathcal{P}$, the goal of the caching policy is to maximize

$$\sum_{p \in \mathcal{P}} \mathrm{util}(\mathrm{sim}(c(p), p)),$$

subject to the IO cost being at most b. Here $\mathrm{sim}(p, q) \in [0, 1]$ denotes the similarity between keys of items p and q and $\mathrm{util}(s)$ denotes the utility of offering an item of similarity s to the requested item. Thus the objective depends on the similarity function $\mathrm{sim}(\cdot, \cdot)$ and the utility function $\mathrm{util}(\cdot)$.

Utility function. The function $\mathrm{util}(\cdot)$ offers a knob to control the aforementioned tradeoff between the similarity of offered and requested items and the IO cost. For instance, if $\mathrm{util}(s) = 1$ for $s = 1$ and 0 otherwise, then our problem formulation reduces to exact caching. Another instantiation of $\mathrm{util}(\cdot)$, which is more relaxed than exact caching, is when $\mathrm{util}(s) = 1$ for $s \geq \tau$ and 0 otherwise, where $\tau \leq 1$ is a given threshold. The resulting objective is called the *threshold objective* and we study this in detail in Section 3.3. In general, $\mathrm{util}(\cdot)$ does not have to be a threshold function; it can be any monotone function. We call this the *smooth objective* and study this in Section 3.4. As we will show later, the threshold and smooth objectives are sufficiently different, and thus it makes sense to study them separately.

3.3 Caching policies: Threshold objective

Let $\tau > 0$ be a given threshold. Recall in the threshold objective we have $\mathrm{util}(s) = 1$ for $s \geq \tau$ and 0 otherwise. From this definition, the hit-or-miss determination is straightforward: a cache hit is said to happen if and only if there is a cached item with similarity at least τ to the requested item. Note that this amounts to a nearest-neighbor search within the cache; in Section 3.5 we will discuss an efficient way to address this problem in practice.

More than one cached item can hit a requested item p; let $\mathcal{C}_\tau(p)$ denote the set of such items. Thus, a cache hit happens iff $\mathcal{C}_\tau(p) \neq \emptyset$. Of all the items in $\mathcal{C}_\tau(p)$, the one most similar to p is offered by the policy, i.e., $c(p) = \arg\max_{q \in \mathcal{C}_\tau(p)} \mathrm{sim}(p, q)$. It is clear that a cached item q can offer hits for all the requests that fall inside a ball $B_\tau(q)$ of radius τ around q. An effective similarity caching policy should judiciously use these balls to "cover" the space of items.

We now generalize the eviction policies in exact caching to work in the similarity case. We begin by generalizing LRU policy as follows: on receiving a request p, first determine if it is a hit or miss by the above definition. If it is a hit, update the reference time of $c(p)$. If it is a miss, bring p to the cache, and evict the cached item with the oldest reference time (as in the vanilla LRU). We refer to this policy as SIM-LRU. The LFU policy can also be generalized in the same way to derive SIM-LFU. Note that both SIM-LRU and SIM-LFU have the following property: no two items in the cache are within similarity τ of each other. In other words, the cache does not contain any redundant items.

3.3.1 Incremental re-clustering

While SIM-LRU and SIM-LFU policies ensure there is no redundant item in the cache, some redundancy can still creep in under these policies. This can happen if the balls around each cached item significantly overlap.

We give an extreme example to illustrate this. Let the cache size be 1 and let τ be a fixed similarity threshold. Let

a, b, c be three points such that $\text{sim}(a, c) = \tau/2$, $\text{sim}(a, b) = \tau$ and $\text{sim}(b, c) = \tau$. Consider the following stream of requests: $a, b, c, b, a, b, c, b, \ldots$. It is easy to see that under the SIM-LRU policy the cache contains either item a or c, and the cache miss rate is asymptotically 1/3. However, if the policy were to cache b, then the miss rate is essentially zero, since b can be used to serve both a and c. This leads to the point we raised before: an astute policy would bring b into the cache even though the current cache item (either a or c) would have offered a hit under similarity. This is the underlying intuition behind our policies described next. We explain our ideas using LRU policy as an example, but the same ideas can be applied to LFU.

First, for each cached item p (i.e., both $\text{key}(p)$ and $\text{val}(p)$ are present in the cache), we store the keys of all the items served by p, i.e., keys of all items that fall in the ball $B = B_\tau(p)$. Each ball has a *representative*, $\text{rep}(B)$, which is initially p. Thus, the cache data structure is a set of balls and for each ball B, the key and value of a representative $\text{rep}(B)$ and a set $\text{hst}(B)$ of keys of past requests served by B.[1] (Since $|\text{key}(p)| \ll |\text{val}(p)|$, storing $\text{hst}(B)$ is inexpensive from a practical point of view.) Our goal is to tightly cluster the items inside the balls and minimize overlaps between the balls to avoid redundancy. We will accomplish this by appropriately updating the representative in a ball.

The policy works as follows: on receiving a request p, we compute the similarity of the request item with the representative of each ball. Say the closest representative is $r = \text{rep}(B)$. If $\text{sim}(r, p) < \tau$, then we fetch p from the disk to serve the request. If $\text{sim}(r, p) \geq \tau$, then we use r to serve the request. We perform the following operation before serving the next request.

We first tentatively add p to $\text{hst}(B)$. We then check if any other item in $\text{hst}(B)$ makes a better representative than the current representative r. To be a new representative, an item $r' \in \text{hst}(B)$ has to satisfy the following two properties:

(1) $\text{sim}(r', p) \geq \tau$ for each $p \in \text{hst}(B)$, and

(2) it has the maximum sum of similarity to all the items in the ball, i.e.,

$$r' = \arg \max_{p \in \text{hst}(B)} \sum_{q \in \text{hst}(B)} \text{sim}(p, q).$$

The first property ensures that r' can be used to serve every item in the ball and the intuition behind the second property is that r' should "cover" the items in its ball as efficiently as possible (i.e., it lies close to the center of the ball).

To update the representative of B, we first compute the total similarity score, as shown above, for every item in B. Note that this computation can be performed incrementally to make its complexity linear in $\text{hst}(B)$. Let r' be the item satisfying (2). Then, one of following two scenarios can arise. If r' also satisfies (1), then we update $\text{rep}(B) = r'$ and fetch $\text{val}(r')$ from the disk (this costs an IO operation). If r' does not satisfy (1), then we delete p from $\text{hst}(B)$, form a new ball around p, with p as its representative; ball B is left unchanged. This also costs an IO operation. We refer to this policy by CLS-LRU.

In our earlier example, it is easy to check that b will be cached by CLS-LRU. In Section 4 we demonstrate its performance improvement over SIM-LRU on a real dataset.

[1] In practice, $\text{hst}(B)$ contains keys of only a subset of past requests served by B for efficiency purposes.

3.4 Smooth objective

So far we have studied similarity caching policies under the threshold objective. In particular, we showed how SIM-LRU policy can allow redundancy in the cache and then proposed CLS-LRU to address this. Next we focus on the smooth objective setting, i.e., $\text{util}(\cdot)$ is any monotone function.

First we discuss how the threshold-based policies of Section 3.3 can fall short for the smooth objective. Observe that under the threshold objective, for a small $\epsilon > 0$, two cache hits with similarities τ and $\tau + \epsilon$ have the same utility, while under the smooth objective they do not. Hence, when a threshold-based policy, say, SIM-LRU, is used for the smooth objective, if a request item p appears and has similarity τ to a cached item, the policy will use this cached item to offer a hit. In doing so the policy incurs a utility loss of $1 - \text{util}(\tau)$. If p appears too many times, this loss can accumulate to become significant enough. On the other hand, an alternative would be to bring p in the cache (incurring an IO cost), if we believe this can avoid the recurring utility loss. Clearly, doing this is profitable only if item p is requested often enough.

Based on the above intuition, we propose our next randomized policy called RND-LRU. (Again, we describe our ideas in terms of LRU, but they apply to LFU as well.) On receiving a request p, the RND-LRU policy finds the cached item, say q, with the highest similarity to p. Then with probability $\alpha(1 - \text{util}(\text{sim}(p, q)))$ the policy declares a cache miss and reads item p from the disk. With the remaining probability it uses q to serve the request. Here $\alpha \in [0, 1]$ is a parameter that controls the utility vs. IO tradeoff.

Observe that if $\text{sim}(p, q)$ is small, then p is likely to be declared a cache miss (as desired). And if $\text{sim}(p, q)$ is very high, then it is likely to be declared a hit. But if p occurs often enough, even if $\text{sim}(p, q)$ is very high, our randomized policy will declare it a cache miss at some point and hence p will make it to the cache. Thus, this simple policy is effective in trading off recurring utility losses vs IO costs. Note that there is little or no bookkeeping in this policy.

Our proposed caching policies (in Sections 3.3 and 3.4) assume each cache item to be of uniform cost and utility. Extending it for non-uniform functions, as done for conventional caching in [5], is a part of our future work.

3.5 LSH realization

As discussed in Section 3.3, a hit-or-miss determination in similarity caching requires to solve a nearest-neighbor problem in high-dimensional space. In other words, given an item p, we need to obtain the cached item q such that $\text{sim}(p, q)$ is maximized, where the similarity function is defined on the space of the keys of the items. To do this efficiently, we use locality sensitive hashing (LSH) [13]: we hash the keys of items using an LSH function with the property that keys are hashed to the same value if and only if they are similar according to $\text{sim}(\cdot, \cdot)$. In our content-match application, we use *weighted Jaccard* measure for the similarity between two vector-valued keys $x = (x_1, \ldots)$ and $y = (y_1, \ldots)$: $\text{sim}(x, y) = \sum_{i:\neg(x_i = y_i = 0)} \min(x_i, y_i)/\max(x_i, y_i)$. It is well-known that min-wise independent hash functions can be used in this case [4].

In Section 4.4.3 we show how LSH significantly reduces the computation cost of hit-or-miss determination while degrading the performance negligibly.

4. EXPERIMENTS

In this section we perform an empirical evaluation of our similarity caching policies. Then in Section 5 we analyze them theoretically.

4.1 Data description and experimental setup

We obtained a page request log from a content-match system. The log contains a time-ordered sequence of more than eight million page requests. For each page request, the system prepares a weighted feature vector taking the page content, site, and user information into account; this feature vector is used to determine the relevant ads for the page. The set of relevant ads are the candidates for the system's final ad selection process, which takes various business and advertisers' constraints into account (of which some can be time-sensitive) in selecting ten or fewer ads that are eventually displayed on the page. To select the relevant ads quickly at page request time, the system extracts features for ads and builds an inverted index to store them. For each feature the index has a *posting list* containing the identifiers of ads that have the feature. To determine relevant ads for a page request, the system probes the index using the WAND algorithm [3], which has been designed to minimize the number of index operations required for answering long queries, as is typically the case for content-match systems.

We simulate a cache on this request log in our experiments. Each item in the cache corresponds to a page request and contains the feature vector of the page request (this vector constitutes the *key* and is usually around one 1KB) and the 1000 most relevant ad candidates for the page (this set of ads constitutes the *value* and is usually in tens of MBs).[2] In our caching policies, to compare the similarity between a new page request and a cached item, we apply the same weighted Jaccard similarity function that is used to match ads by the content-match system.

When a page is requested, we perform a look-up operation in the cache to search for it. If the page is not found (i.e., a miss occurs), we probe the index to find the 1000 most relevant ad candidates for the page. (In case of a hit, we get these candidates from the cache.) Then, the final ad selection process is run on these candidates. Hence, a cache hit saves us from probing the index, which is an expensive operation, in terms of time, because it requires accessing data from disk and seeking and examining hundreds of posting lists on average.

4.2 Tolerance to similarity

In this experiment we demonstrate why similarity caching is well suited for such content-match systems. In particular, we show that the relevant ads for similar page requests have significant overlap. Hence, on receiving a new page request, if we find a page request in the cache that is similar to it, the top 1000 relevant ads for this cached request can be used to serve the new request as well. By doing so the content-match system can avoid probing the ad index and save a lot of computation/IO cost.

For this experiment we sampled 500K pairs of page requests from the request log. Then for each pair, say $\langle p, q \rangle$, we computed the similarity in page requests ($\text{sim}(p, q)$) and the Jaccard overlap in the top 1000 ads for them ($\text{ads}(p, q)$).

[2]Note that two different cache items can have overlapping ads. This causes redundancy and can be avoided by buiding an in-cache index; we leave this as part of our future work.

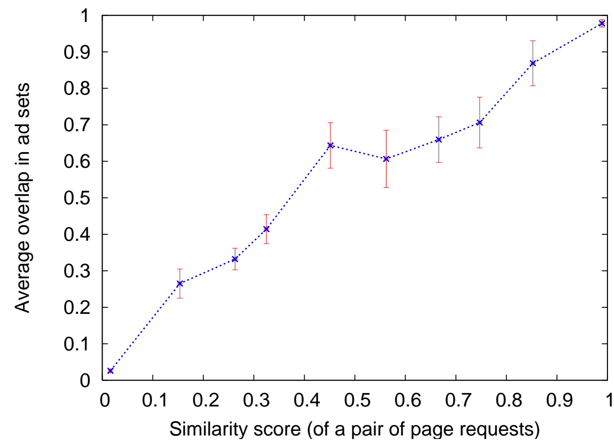

Figure 1: Relationship between similarity in page requests and Jaccard overlap of the sets of ads chosen for those requests.

Pairs with similar values of $\text{sim}(p, q)$ are put together in a bucket. In Figure 1 we show the result. Each point in the figure denotes a bucket where the x-coordinate is the average similarity score $\text{sim}(p, q)$ and the y-coordinate is the average Jaccard overlap $\text{ads}(p, q)$ of the pairs in the bucket. As it is clear from the figure, as the similarity score between two page requests increases, the overlap in the sets of relevant ads for them also increases.

4.3 Feasibility

In this experiment we show the advantage of similarity caching over exact caching in general (and not under any specific caching policy). We take the first 100K items from the request log and ask the following question: given a similarity threshold what is the minimum size of the cache such that all 100K page requests are covered by the items in the cache. (The cache can contain items from this 100K log only.) This is the well-known k-center problem that is NP-hard [10]. Hence, we use the farthest point heuristic, an approximation algorithm known for the k-center problem [11]. Under this heuristic, the cache is constructed one by one. We start with a random item in the cache. Then at each step, we select that item from the 100K items that has the minimum maximum similarity to the existing items in the cache and put it in the cache. The algorithm stops when the minimum maximum similarity drops below the given threshold.

As shown in Figure 2, the required cache size increases as the similarity threshold τ increases ($\tau = 1$ implies exact caching). For instance, the required cache size is 80K when $\tau = 1$, but it drops by almost 50% to 40K when $\tau = 0.8$. This shows that similarity caching allows a cache to offer a significant more number of hits in comparison to exact caching, independent of the caching policy. Next we perform this comparison under specific caching policies.

4.4 Threshold objective

First we study threshold objective from Section 3.3.

4.4.1 Performance Comparison

Figure 3 shows the performance of SIM-LRU and SIM-LFU

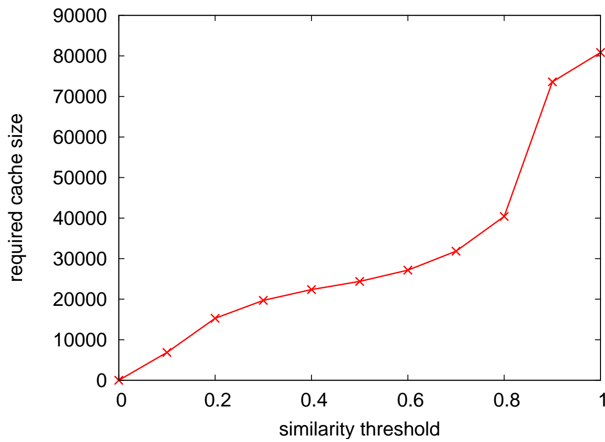

Figure 2: Given a similarity threshold, the minimum size of the cache required to "cover" all 100K pages.

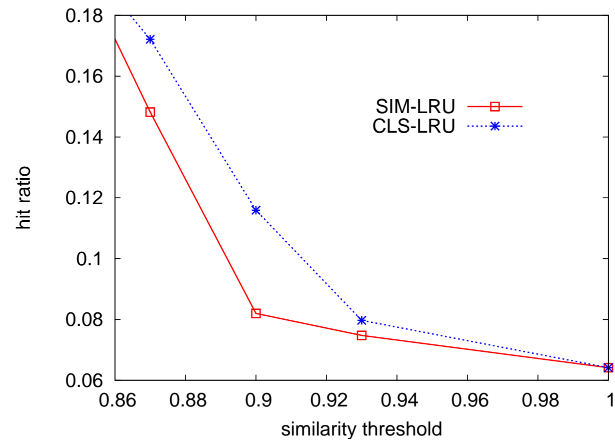

Figure 4: Performance of SIM-LRU policy with and without re-clustering on a cache of 1K size.

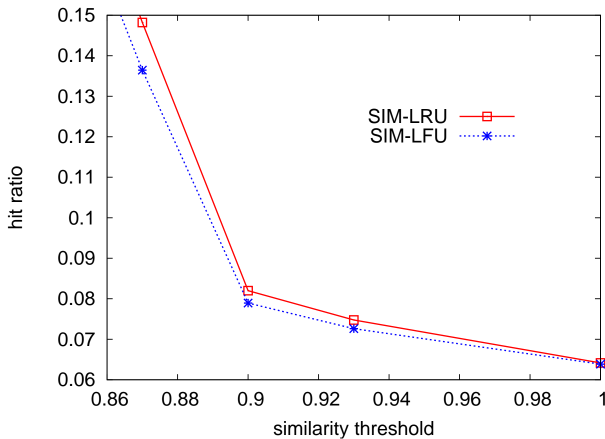

Figure 3: Performance of SIM-LRU and SIM-LFU policies on a cache of 1K size.

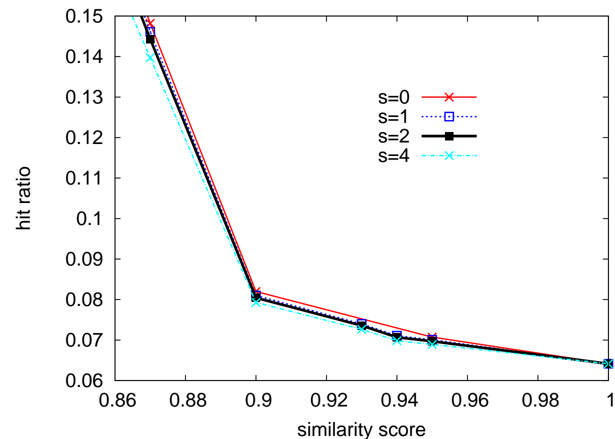

Figure 5: Performance of SIM-LRU policy while using LSH for probing the cache, where s denotes the number of hash functions used in LSH. (The size of the cache is kept to 1K.)

policies for a 1K cache. The x-axis denotes the similarity threshold τ while the y-axis denotes the hit ratio. Note that the hit ratio for SIM-LRU is roughly 7% for $\tau = 1$ (when SIM-LRU reduces to exact LRU policy), but it gets more than doubled when τ is set to 0.88. This shows how a little sacrifice in similarity can dramatically increase the performance of the caching policies.

Both SIM-LRU and SIM-LFU perform similarly in this experiment, so for brevity we explain the rest of the experiments in the context of LRU only. Note that the focus of this paper is on studying similarity caching and not on finding the best exact caching policy.

4.4.2 *Effect of incremental re-clustering*

In Figure 4 we plot the SIM-LRU and CLS-LRU policies. The size of the cache is kept to 1K. Recall that CLS-LRU policy keeps the cached items tightly clustered together and avoids redundancy in the cache. Hence, it is not surprising to see that it performs better than SIM-LRU. As threshold τ gets closer to 1, both policies start performing similar and approach the exact LRU policy.

4.4.3 *Effect of using LSH*

Next we measure how LSH affects the performance and computation cost of our caching policies when it is used for hit-or-miss determination (as described in 3.5). Figure 5 shows the performance of the SIM-LRU policy under LSH for a cache of $1K$ size. Here, s denotes the number of hashes used in LSH ($s = 0$ implies no LSH). It is clear that the performance degradation due to LSH is negligible.

In Figure 6 we show the computation cost of hit-or-miss determination. In particular, we plot the average number of similarity score computations performed per request in hit-or-miss determination under SIM-LRU policy. When LSH is not the used ($s = 0$), the policy does an exhaustive search of the cache and thus, incurs a 1K computation cost per request. Observe how the computation cost goes down by a order of magnitude when LSH is used, for a negligible degradation in performance.

4.4.4 *Effect of cache size*

Now that we have confirmed that LSH does not signifi-

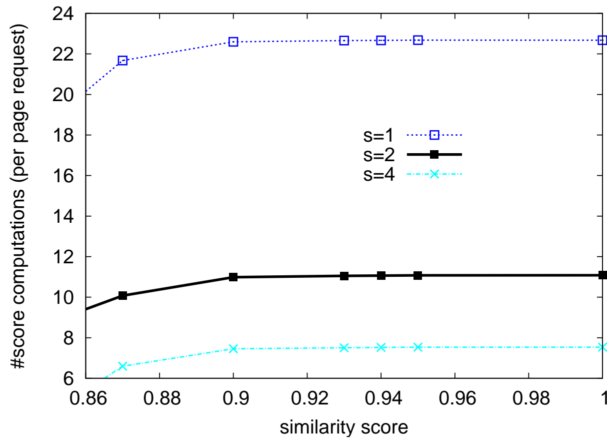

Figure 6: Number of similarity score computations performed per page request, on average, in hit-or-miss determination in SIM-LRU policy (for a 1K cache). s denotes the number of hash functions used in LSH. When LSH is not used (s=0), due to an exhaustive search of the cache the number of score computations per request is 1000.

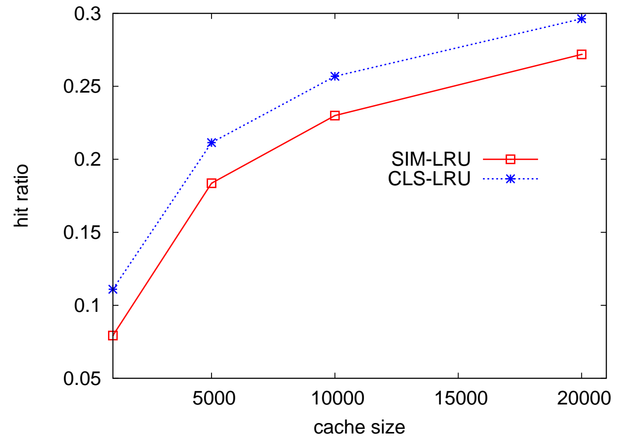

Figure 7: Performance of SIM-LRU policy with and without re-clustering for different cache sizes for similarity threshold $\tau = 0.9$. LSH is used to control the computation cost ($s = 4$).

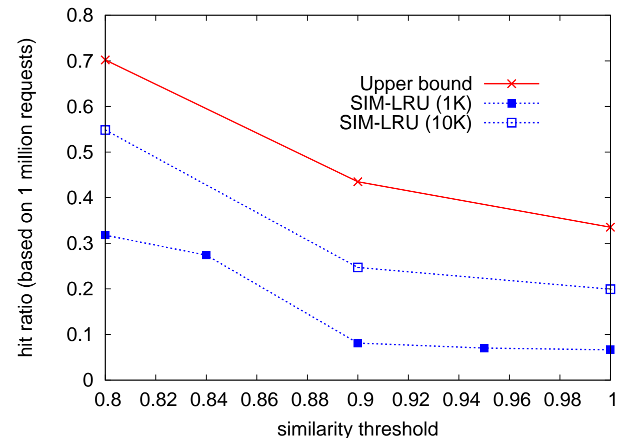

Figure 8: Performance comparison of our policies with the upper bound.

cantly affect the performance, we use LSH to test our policies on larger cache sizes. In Figure 7 we show the performance of SIM-LRU and CLS-LRU policies for different cache sizes. The number of hashes used in LSH, s, is set to 4 and similarity threshold τ is set to 0.9. Similar to Figure 4 CLS-LRU continues to perform better than SIM-LRU.

It is tempting to think that for large cache sizes, the cache can allow some redundancy and thus re-clustering may not be needed. However, from Figure 7 we note that re-clustering is considerably beneficial even for large cache sizes.

To illustrate how exact caching is not effective in our scenario, we performed experiments to figure out that exact caching requires a cache size of more than 100K items to achieve the hit ratio of 30% (achieved by our CLS-LRU policy using a 20K cache). Assuming a cache item to occupy 10MB of storage, a 100K-item cache requires 1TB of memory and is beyond the budget of most content-match systems.

4.4.5 Performance comparison with an upper bound

In this experiment we compare the performance of our policies with an upper bound. This allows us to see how good our policies are on the absolute scale. We obtain the upper bound in the following way: for a request we check if the request has more than τ similarity to any earlier request in the log. If it has, then we call this request a hit, otherwise it is called a miss. Clearly, no caching policy can do better than this (without using any extraneous information).

Figure 8 shows the upper-bound and the performance of SIM-LRU policy on a 1K and 10K size cache. As it is clear from the figure that our policies come quite close to this upper bound for a fairly modest cache size.

4.5 Smooth objective

Next we study the smooth objective (of Section 3.4). Recall from Section 3.2 that smooth objective involves a function util($\cdot$) that gives the utility of a hit for a similarity value. For our content-match application, we can use Figure 1 to derive this function. In the figure we showed the relationship between the similarity of two page requests and the overlap in their sets of relevant ads. If we count utility in terms of this overlap, then a linear or a quadratic form of util($\cdot$) function appears meaningful.

Figure 9 shows the performance of SIM-LRU and RND-LRU policies for linear (i.e., util(s) = s) and quadratic (i.e., util(s) = s^2) utility functions. The x-axis denotes cost in terms of the fraction of page requests which were not answered from the cache (and required probing the ad index). The y-axis is the average utility per request. SIM-LRU has the threshold τ parameter while RND-LRU has the α parameter to control the tradeoff between cost and utility.

Observe that both policies perform fairly similar under high cost budgets, but when the cost is set low, RND-LRU performs significantly better than SIM-LRU. The reason behind it is that when cost is small, SIM-LRU policy operates under a loose similarity threshold τ and offers hit using the cached items that are quite far from the requested items. In RND-LRU some of these requests get declared as cache

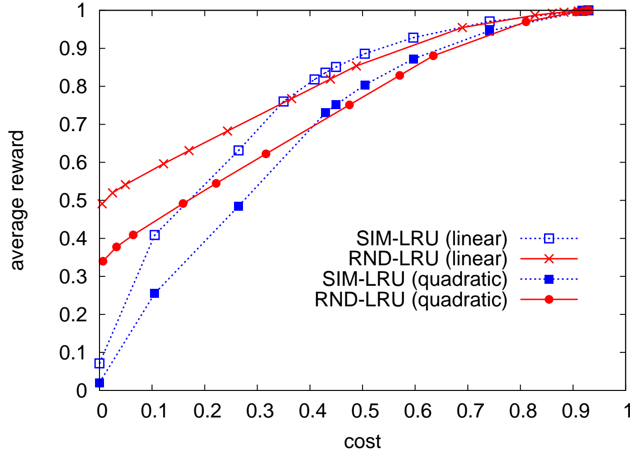

Figure 9: Performance of SIM-LRU and RND-LRU policies for 1K cache for different cost budgets under linear and quadratic utility functions.

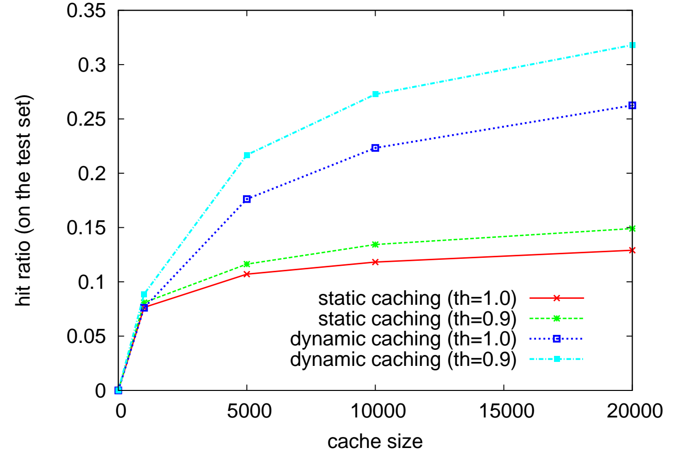

Figure 10: Static vs. dynamic policies for exact and similarity caching.

misses when they appear often enough, and hence they are brought from disk to the cache. This leads to a better covering of the high-dimensional space of items, which results in a higher utility in turn.

4.6 Static caching

In this experiment we compare the performance of our caching policies with *static caching* wherein the cache is not updated over time. Typically, a static cache is constructed by mining previous request logs and keeping the most frequent items in the cache to hit as many future requests as possible. Previous studies have shown that for certain Web applications static caching works well [2, 20], thereby motivating us to compare it with dynamic caching. The comparison is done for different similarity thresholds.

For this experiment we took the first 2 million requests from the log. We put the first 250K of these requests in the training set, while the rest were put into the test set. For constructing a static cache given a similarity threshold, we employ the k-center algorithm on the training set and put the most populated k-centers in the cache. This cache is then used to serve the test set (and not updated at any time). Figure 10 compares the performance of static caching with dynamic caching (SIM-LRU to be specific) on the test set. We perform the comparison for similarity threshold τ equal to 1 (which refers to exact caching) and $\tau = 0.9$.

As expected, both static and dynamic caching perform better when the similarity threshold τ is lowered. More importantly, we see that dynamic caching performs much better than static caching. This shows that page requests for our content-match application have strong temporal patterns and dynamic caching is able to exploit that, unlike its counterpart which remains static, by definition.

5. A GENERATIVE MODEL

To explain the performance boost added by similarity caching, we propose a simple generative model for the data and then prove a lower bound on the performance of similarity caching vis-a-vis exact caching. For simplicity, we deal with distances instead of similarities, where $d(p, q) = 1 - \mathrm{sim}(p, q)$; we assume $d(\cdot, \cdot)$ is a metric.

Intuitively, the model can be described as *noisy copying*: with some probability each new arriving item is a fuzzy copy of one of the previous items. Formally, the model is parametrized by four values: p, α, β, and k. When a new item q arrives, we look at the past k items. With probability p, the item q represents a perturbed copy of one of the previous k cached items. More specifically, with probability $p\beta$, there is no perturbation, and q is an exact copy. And with probability $p(1 - \beta)$, q is selected so that it is at distance d from a previous item, where d is distributed as $N(0, \alpha^2) \cap (0, 1]$;[3] this is the perturbed copy. Finally, with probability $1 - p$, q is generated uniformly at random in space, i.e., the distance between q and any of the past k items is uniform in $[0, 1]$.

The added advantage of similarity caching can be traced exactly to the fuzzy copy step. For exact caching to perform well, we need β to be relatively high, otherwise the chance for a new item to be a cache hit would be prohibitively low. Similarity caching, on the other hand, takes advantage of the perturbed copies — if the value of α is positive, then some of the incoming items would fall within the threshold afforded by similarity caching. In fact, we can show formally the improvement on the hit rate due to similarity caching.

To validate the model, and test the degree of perturbed copies in our data, we ran a maximum likelihood test to obtain the best values for p, α, and β in the dataset. For $k = 1000$, the best fit occurred at: $p = 0.18$, $\alpha = 0.2$, and $\beta = 0.4$. The exact copying parameter, β is relatively high, so it is not surprising that exact caching does yield a noticeable improvement in performance over no caching at all. However, observe that the α parameter shows that there is room for similarity caching. That is, of the items that are perturbed copies, more than 65% (one standard deviation) lie within a distance of 0.2 (similarity ≥ 0.8). And it is these items that provide the extra performance boost over exact caching.

[3] That is, a gaussian of mean 0, variance α^2, limited to the interval $(0, 1]$; the pdf $f(x)$ of this variable is $f(x) = \sqrt{\frac{2}{\pi}} \cdot \exp\left(-\frac{x^2}{2\alpha^2}\right) / \left(\alpha \cdot \mathrm{erf}\left(\frac{1}{\alpha\sqrt{2}}\right)\right)$, for $x \in (0, 1]$ and $f(x) = 0$ otherwise.

5.1 Analysis

Given the model above, we proceed to provide a bound on the extra performance afforded by similarity caching. While the algorithms we analyze are not exactly those we experimented with, in some sense they represent the best possible scenario for exact caching and one of the worst scenarios for similarity caching. The added benefit provided by real, as opposed to stylized (yet analyzable) algorithms only serves to increase the gap between exact and similarity caching.

An equivalent model. It is easier to consider the following equivalent model, represented by a weighted tree, with items positioned at its nodes. The (transitive closure of the) weighted tree will be our metric. The model is as follows. At time $t = 1$, we have a tree with a single node and an item on the node. At time $t > 1$, an item x is chosen arbitrarily among those generated in the last k steps. Let N be the node where x was placed. Then, with probability $p\beta$, the new item y is placed on N; otherwise a new node N' is added to tree as a child of N and the new item y is placed on N'. The weight d of the edge (N, N') is chosen uniformly at random in $[0, 1]$ with probability $1-p$ and from $N(0, \alpha) \cap (0, 1]$, with probability $p(1-\beta)$.

We first obtain a simple upper bound on the performance of exact caching.

Theorem 1. *The probability that exact caching incurs a cache hit is* $\leq p\beta$.

Proof. Suppose that the new item y is about to be generated, using item x. Each of the items in the cache $c_1, \ldots, c_t$ will be at some distance to x, say $d(x, c_i) = d_i$ with $d_1 \leq d_2 \leq \cdots \leq d_t$. Let j be the number of d_i's equal to 0, $j = |\{i \mid d_i = 0\}|$. The probability that c_i can be used to answer y, if $d(c_i, x) > 0$, is 0 under exact caching. On the other hand, note that the c_i's having $d_i = 0$ are all equal to each other. The probability that those can be used to answer x is $p\beta$. $\square$

Let us now consider the following threshold objective similarity caching policy, parametrized by distance threshold $\delta \in (0, 1]$ (i.e., $\delta = 1 - \tau$ where τ is the similarity threshold). For analysis purposes, we will assume the eviction policy to be a slight modification of FIFO (instead of LRU or LFU).

(1) The cache will store keys of the last k items, together with the items used to return the answer. That is, for $i = 1, \ldots, k$, the i-th entry of the cache will contain the pair (x_i, x'_i) where x_i is the key of the requested item and x'_i is the answer we returned for this request; (we will always have $d(x_i, x'_i) \leq \delta$). We apply a FIFO eviction policy: when a new item y comes in, we evict the oldest pair and add a new pair (y, y') to the cache, where y' is either x'_i for some i (when we have a cache hit) or a newly computed answer y' (when we have a miss).

(2) To determine hit-or-miss when a new item y arrives, we search the cache for the closest cache item to $y : x_{i*} = \arg\min_{x_i} d(x_i, y)$. Then,

a. if $(d(y, x_{i*}) > 0$ and $d(x_{i*}, x'_{i*}) > \frac{\delta}{2})$ or $(d(y, x_{i*}) > \frac{\delta}{2})$ or $(d(y, x'_{i*}) > \delta)$, we fault (computing the exact answer for y) and add (y, y) to the cache,

b. else we add (y, x'_{i*}) to the cache,

(3) Finally, if we faulted and computed the exact answer for y, we try to "improve" each cache pair using y; that is, we update a pair (x_i, x'_i) to (x_i, y') when $d(x_i, x'_i) \geq d(x_i, y')$.

Note that in this algorithm we may choose to fault even when we did not have to, for example when the cached item and its answer are at a distance of more than $\delta/2$ away. Even with these extra faults, we show that the hit ratio of this algorithm is much better than that of exact caching.

Theorem 2. *Suppose the cache has the same size k as the model. Then, the hit ratio of* SIM-FIFO *with distance threshold $\delta \in (0, 1]$, is*

$$\geq (1 - o(1)) \left(p\beta + (1-p)\frac{\delta}{4} + p(1-\beta)\frac{\operatorname{erf}\left(\frac{\delta}{\alpha 2^{3/2}}\right)}{2 \cdot \operatorname{erf}\left(\frac{1}{\alpha 2^{1/2}}\right)} \right).$$

Proof. Recall that for each cache item (x_i, x'_i) we have $d(x, x'_i) \leq \delta$. We call an item *good* if $d(x_i, x'_i) \leq \frac{\delta}{2}$ and *bad* otherwise.

Let ξ_e (exact) be the event that the distance generated by our generative model is 0, ξ_c (close) be the event that the distance generated by our model is in $\left(0, \frac{\delta}{2}\right]$, and ξ_f (far) be the event that the distance is in $\left(\frac{\delta}{2}, 1\right]$. Let q_e, q_c and q_f be the probability with which these events occur.

Then, $q_e = p\beta$, $q_f = 1 - q_c - q_e$, and

$$q_c = \int_0^{\delta/2} ((1-p) + p(1-\beta)f(x))\, dx$$

$$= (1-p)\frac{\delta}{2} + p(1-\beta)\frac{\operatorname{erf}\left(\frac{\delta}{\alpha 2^{3/2}}\right)}{\operatorname{erf}\left(\frac{1}{\alpha 2^{1/2}}\right)}.$$

Suppose the adversary chose an item x_i (from the last k items) to generate the new item y, as dictated by our generative model. Note that due to the FIFO eviction policy, (x_i, x'_i) must be present in the cache. Since d is a tree metric, the closest item to y will be x_i. If x_i is a good element (i.e., $d(x_i, x'_i) \leq \frac{\delta}{2}$), then:

- with probability q_e item y is an exact copy of x_i. We can answer $y = x'_i$ without faulting, and add a good item, (y, x'_i), to the cache.
- with probability q_c we have a cache hit because $d(y, x'_i) \leq d(y, x_i) + d(x_i, x'_i) \leq \delta$. The new pair (y, x'_i) may be either good or bad.
- with probability q_f we will fault, and the new pair (y, y) must be good.

On the other hand, suppose that the adversary chose an item x_i, that is a bad element to generate the new item y. Then,

- with probability q_e, we have a cache hit. And we add a bad item (y, x'_i) to the cache.
- with probability q_c we fault, bringing the good element (y, y) into the cache. In the update phase since the distance $(x_i, x'_i) \geq \frac{\delta}{2}$ and $d(x_i, y) \leq \frac{\delta}{2}$ we update (x_i, x'_i) to (x_i, y), thereby decreasing the number of bad elements. (Pair (x_i, y) may get evicted afterwards by the FIFO policy if x_i is the least recently used element).
- with probability q_f we fault and add a new good element (y, y) to the cache.

Note that if the adversary chooses a good element we have a cache hit with probability at least $q_e + q_c$. On the other hand, if the adversary chooses a bad element, we can lower bound the probability of a cache hit with q_e. Fix a cache at time $t = 0$, and a sequence of length ℓ. Consider the queries in the ℓ-sequence generated by ξ_c event: let B be the fraction of these queries generated from a bad item. It is easy to see that the overall hit rate will be $q_e + q_c(1 - B)$. We will show that $\mathbf{E}[B] \leq (1 + o(1))\frac{1}{2}$, thereby completing the proof.

Let X_t the number of *distinct* elements that are bad that are still in the cache at time t, or have been evicted at some time $t' : t_0 \leq t' \leq t$. Note that as the sequence progresses, X_t increases ($X_t = X_{t-1}+1$) iff the adversary chooses a *good* item and event ξ_c happens, and X_t decreases ($X_t = X_{t-1} - 1$) iff the adversary chooses a *bad* item and ξ_c happens.[4] In case ξ_e happens, the new item is an exact copy and the number of *distinct* bad items stays the same; in case of ξ_f the new item is good.

Note that the expected number of changes to X_t after a sequence of length ℓ is exactly $\ell \cdot q_c$. Furthermore, since X_t can never become negative, the total number of decreases is at most the total number of increases plus k (that is, the k original cache items, that could all have been *bad*). Since k is an additive factor, as $\ell \to \infty$, the total number of decreases (and thus the total number of *bad* adversarial choices that triggered a ξ_c event) makes up at most a $B = (1 + o(1))\frac{1}{2}$ fraction of all X_t's changes, which implies that on almost half of the ξ_c events the algorithm will not fault. □

Discussion. Recall, that the best fit for the model with $k = 1000$ was $p = 0.18, \alpha = 0.2$ and $\beta = 0.4$. Our analysis tells us that the exact hit rate should be less than 7% and the similarity caching hit rate with $\tau = 0.8$, must be at least 14%. Experimentally, the hit rate was 6.4% for exact caching and 24% for similarity caching. This is a very close match to the model results, especially taking the fact that the similarity caching algorithm is much more advanced than the FIFO algorithm analyzed.

6. CONCLUSIONS

We formally defined the problem of similarity caching in this paper and studied it under two different objectives, i.e., threshold and smooth. Our proposed caching policies incrementally re-organize the cache to ensure that the cached items cover the similarity space efficiently. By conducting extensive experiments on real data we demonstrated how similarity caching can significantly improve the performance of content-match systems, without compromising much in terms of accuracy.

Supplementing our empirical evaluation, we proposed a simple generative model of content-match page requests. We validated this model by fitting it to the real data and provided a theoretical analysis of similarity caching under this model.

7. ACKNOWLEDGMENTS

We would like to thank Arun Iyengar for his insightful comments, and Jason Zien for his help with the data.

[4]Note that if the chosen bad item is also the least recently used element in the cache, then it is converted into a good item before eviction by our policy.

8. REFERENCES

[1] M. F. Arlitt and C. L. Williamson. Trace-driven simulation of document caching strategies for internet web servers. *Simulation J.*, 68:23–33, 1997.

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[3] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-Level retrieval process. In *Proc. 12th CIKM*, pages 426–434, 2003.

[4] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *JCSS*, 60:630–659, 2000.

[5] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[6] F. Chierichetti, R. Kumar, and S. Vassilvitskii. Similarity caching, 2008. Manuscript.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. 20th SoCG*, pages 253–262, 2004.

[8] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM TOIS*, 24(1):51–78, 2006.

[9] F. Falchi, C. Lucchese, S. Orlando, R. Perego, and F. Rabitti. A metric cache for similarity search. In *Proc. 6th Workshop on LSDSIR*, 2008.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., 1979.

[11] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *TCS*, 38(2-3):293–306, 1985.

[12] P. Indyk. Low-distortion embeddings of finite metric spaces. In *Handbook of Discrete and Computational Geometry.* CRC Press, 2004.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th STOC*, pages 604–613, 1998.

[14] R. Lempel and S. Moran. Optimizing result prefetching in Web search engines with segmented indices. In *Proc. 28th VLDB*, pages 370–381, 2002.

[15] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. 12th WWW*, pages 19–28, 2003.

[16] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. 14th WWW*, pages 257–266, 2005.

[17] E. P. Markatos. On caching search engine query results. In *Computer Communications*, pages 137–143, 2000.

[18] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[19] P. C. Saraiva, E. S. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. 24th SIGIR*, pages 51–58, 2001.

[20] I. Tatarinov, A. Rousskov, and V. Soloviev. Static caching in Web servers. In *Proc. 6th CCN*, page 410, 1997.

[21] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. 17th WWW*, pages 387–396, 2008.